Projet de Compléments en Programmation Orientée Objet n° 1 : Automates cellulaires

Ce projet est à réaliser en binôme et est à rendre via la section « Travaux » de la plateforme DidEL avant le 9 novembre. Prévoyez une courte documentation, expliquant notamment au correcteur ce qu'il doit s'attendre à trouver dans votre projet et comment il doit l'exécuter.

I) Automates cellulaires

Un automate cellulaire est la donnée d'une grille régulière, sur laquelle « vivent » des cellules : des automates à états finis tous identiques, dont la fonction de transition dépend de l'état courant de la cellule et de celui des cellules voisines (la notion de voisin dépendant du type de grille).

En général on se donne aussi une configuration initiale (une fonction de la grille vers les états de cellule) et on regarde comment le système évolue pour chaque configuration initiale.

Le but de ce projet est d'écrire un simulateur d'automates cellulaires suffisamment générique pour permettre d'implémenter *facilement* plusieurs variantes de ces automates, en s'inspirant de la liste ci-dessous ou trouvés sur internet (point de départ possible : page « automate cellulaire » de Wikipédia).

II) Exemples

Voici une liste d'exemples. Bien qu'il ne soit pas demandé de tous les implémenter, déterminez déjà quel exemple est cas particulier de quel autre pour faire un schéma de l'héritage.

Jeu de la vie L'exemple d'automate cellulaire le plus connu est celui du jeu de la vie de Conway. Dans cet automate, la topologie est celle d'une grille infinie classique (à maillage carré) avec la relation de voisinage usuelle (8 voisins : les 4 directions principales plus les diagonales).

Une cellule peut avoir deux états : vivant ou mort. L'état suivant dépend du nombre de voisins en vie :

— si la cellule est vivante, elle se maintient en vie si elle a 2 ou 3 voisins en vie (sinon elle meurt d'isolement ou, au contraire, d'étouffement) ;

— si la cellule est morte, la cellule devient vivante si elle a exactement 3 voisins en vie.

Variante : comme il est compliqué de simuler une grille infinie, on préfère souvent considérer une grille « torique » : sur la première ligne, une cellule a comme voisin du haut la cellule de la même colonne située sur la dernière ligne (idem en invertissant « premier » et « dernier », en remplaçant « haut » par « bas » ; idem en intervertissant ligne et colonne, remplaçant « haut » par « gauche », et ainsi de suite).

Variantes Il existe de nombreuses variantes du jeu de la vie. Par exemple, *HighLife*, *Day&Night* (voir Wikipédia) ou Fredkin (voir Google) sont également basé sur des cellules à deux états (mort ou vivant) mais des règles différente pour naître, survivre ou mourir.

Automate cellulaire mort/vivant isotrope L'état d'une cellule dépend de ses voisins, mais pas de leurs positions. Un tel automate est défini par deux sous-ensembles de $\{0, 1, \ldots, \#\text{voisins}\}$, *survivre* et *naître* :

— Si une cellule est vivante, elle reste vivante si le nombre de ses voisins vivants appartient à l'ensemble *survivre* ; sinon elle meurt.

— Si une cellule est morte, elle devient vivante si le nombre de ses voisins vivants appartient à l'ensemble *naître* ; sinon elle reste morte.

Plus d'états Par exemple, la variante *Immigration* considère des cellules à trois états : deux états *vivant* et un état *mort*. Les règles sont les même que pour le jeu de la vie, si ce n'est qu'une cellule naissante prend l'état *vivant* majoritaire parmi ses voisins.

Grille finie Jusqu'à maintenant, on a considéré qu'un type de grille « torique ». Or, dans tous les cas précédents, il y a un état par défaut : l'état *mort*. Dans ce genre de cas, on peut considérer une grille finie non-torique : certaines cases ont des voisins inexistants qui sont considéré comme ayant l'état *par défaut*.

Règle ne dépendant pas du nombre d'états On considère n états $1, 2, \ldots, n$. L'automate de Griffeath à n états possède la règle suivante :

— Une cellule dont l'état est i passe à $(i + 1)$ si au moins 3 voisins ont le même état $(i + 1)$. (Si la cellule est à l'état n, elle passe à l'état 1)

Pour utiliser une grille finie, on peut par exemple rajouter un état par défaut 0 qui n'est jamais atteint par une cellule de la grille. Cet état pourrait-être un joker : une cellule qui touche 3 joker (donc sur un bord) n'a besoin que de deux voisins pour changer d'état et une cellule qui touche 5 joker (un coin) n'a besoin que d'un voisin pour changer d'état.

Pseudo-automate de Griffeath On peut également considérer que le nombre d'états est infini (modélisée par un entier) tout en suivant la règle de l'automate de Griffeath classique.

Topologie de la grille Jusqu'à maintenant, on n'a considéré que des grilles à maillage carré (chaque cellule a 8 voisins, sauf sur les bords éventuellement). À la place, on pourrait considérer des grilles, dont les cellules sont des triangles (3 voisins), et dont l'état dépend de leurs trois voisins, ou bien des grilles dont les cellules sont des hexagones (6 voisins).

On peut également considérer les automates cellulaires à une dimension (les cellules ont un voisin gauche et un voisin droit) : la grille est juste une ligne. Dans ce cas on affiche les générations l'une en dessous de l'autre : la deuxième dimension du plan est le temps. Voir à ce sujet la *règle 126* qui dessine le triangle de Sierpinski ou la *règle* qui donne un comportement chaotique.

Pseudo-automate à moyenne L'état d'une cellule est maintenant un nombre réel entre 0 et 1. À l'étape suivante, il devient la moyenne de ses voisins. Cette règle ne dépend évidemment pas de la topologie de la grille.

Petite variante L'état d'une cellule devient $(x \times y + c)$ où x est la moyenne de ses voisins, y est son ancien état et

— c vaut $0,5$ si $(x \times y) < 0,25$; — c vaut $-0,5$ si $(x \times y) > 0,75$; — c vaut 0 sinon.

On peut étendre encore d'avantage les types d'états (nombres complexes), les types de grilles (pavage régulier pentagone-hexagone) ou les voisinages. La limite est votre imagination.

III) Philosophie de l'implémentation demandée

On veut une implémentation souple, où l'on pourrait facilement remplacer juste la topologie de la grille, ou bien juste le modèle de la cellule. Pour cela, remarquons bien d'abord que :

— La cellule n'a pas besoin de connaître la forme générale de la grille pour fonctionner. Ce qui la caractérise, c'est son espace d'états et sa fonction de transition qui, elle, a juste besoin d'accéder à un certain nombre de voisins (dans un certain ordre).

— La grille n'a pas besoin de savoir comment une cellule est faite. Tout ce qu'elle doit faire, c'est être de capable de dire à une cellule quels sont ses voisins.

Pour résumer, pour qu'une classe de cellules fonctionne avec une certaine classe de grille, il faut et il suffit qu'elles s'accordent sur le type du voisinage.

On vous suggère de vous baser sur les interfaces suivantes :

```
1  public interface State {
2        /**
3         * icône vide par défaut
4         */
5        Icon DEFAULT_ICON = new ImageIcon();
6
7        /**
8         * @return caractère représentant l'état
9         */
10       default char toChar() { return '.'; }
11
12       /**
13        * @return icône représentant l'état
14        */
15       default Icon toIcon() { return DEFAULT_ICON; }
16 }
```

```
1  public interface Cell<S extends State, N extends Enum<N>> {
2        /**
3         * @return état courant de la cellule
4         */
5        S getState();
6
7        /**
8         * Changer l'état de la cellule.
9         * @param state : nouvel état de la cellule.
10        */
11       void setState(S state);
12
13       /**
14        * @param direction : direction du voisin que l'on cherche.
15        * @return référence vers la cellule voisine dans la direction
              désirée.
16        */
17       Cell<S, N> getNeighbor(N direction);
18
19       /**
20        * Simuler une transition, sans changer l'état de l'objet.
21        * @return le prochain état de la cellule après avoir effectué une
```

```
22          *       transition.
23          */
24         S nextState();
25 }
```

```
1  public interface Grid<S extends State, N extends Enum<N>, C extends Cell<?, N>> {
2         /**
3          * Exécuter une transition de l'automate cellulaire.
4          */
5         void update();
6         /**
7          * @return Chaîne représentant l'état de la grille, en vue de
               l'affichage.
8          */
9         String stateAsString();
10 }
```

Typiquement une implémentation `CellImpl` de `Cell<SImpl, NImpl>` dispose d'un attribut permettant de retrouver pour chaque direction de voisinage, le voisin correspondant (peut être de type `Map<NImpl, CellImpl>`, ou bien un objet avec une méthode de `NImpl` vers `CellImpl`), la méthode `getNeighbor()` devra utiliser cet attribut.

Cela peut bien sûr adapté selon vos besoins. Quand c'est possible, préférez étendre ces interfaces en ajoutant ce qui vous manque dans les nouvelles interfaces, plutôt que de modifier directement celles qui sont données.

IV) À faire

1. Implémentez le jeu de la vie (variante torique). L'affichage se fera au moins sur la console en mode texte avec un mode animation continue et un mode pas-à-pas (mode graphique optionnel). Votre programme devra permettre à l'utilisateur de définir l'état initial à partir duquel il souhaite lancer la simulation.

   Ici on utilisera, comme type pour le voisinage (paramètre `N`), l'énumération :

   ```
   1 enum SquareGridNbh {
   2   NORTH, NORTH_EAST, EAST, SOUTH_EAST, SOUTH, SOUTH_WEST, WEST, NORTH_WEST
   3 }
   ```

   et comme type pour l'état (paramètre `S`) :

   ```
   1 enum LifeState implements State {
   2   DEAD { public char toChar{} { return '.'; }},
   3   ALIVE { public char toChar() { return 'O'; }}
   4 }
   ```

2. Implémentez quelques autres types d'automates cellulaires en vous inspirant de la section « Exemples ». L'accent doit être mis sur le fait de réutiliser le plus possible les mêmes composants : on doit pouvoir combiner n'importe quelle grille avec n'importe quel modèle de cellule qui fonctionne avec ce type de voisinage.

   Utilisez autant que possible les interfaces proposées. Ajoutez les fonctionnalités manquantes via l'héritage (de classe et/ou d'interface).

Dans tous les cas, prévoyez des jeux de tests, notamment une bibliothèque de configurations initiales que l'on pourrait charger soit depuis la ligne de commande, soit depuis un menu, ainsi que la possibilité de générer une configuration initiale aléatoirement.